A QoS-Aware Middleware for Fault Tolerant Web Services

Zibin Zheng and Michael R. Lyu
Department of Computer Science and Engineering
The Chinese University of Hong Kong
Hong Kong, China
{zbzheng, lyu}@cse.cuhk.edu.hk

Abstract

Reliability is a key issue of the Service-Oriented Architecture (SOA) which is widely employed in critical domains such as e-commerce and e-government. Redundancy-based fault tolerance strategies are usually employed for building reliable SOA on top of unreliable remote Web services. Based on the idea of user-collaboration, this paper proposes a QoS-aware middleware for fault tolerant Web services. Based on this middleware, service-oriented applications can dynamically adjust their optimal fault tolerance strategy to achieve good service reliability as well as good overall performance. A dynamic fault tolerance replication strategy is designed and evaluated. Experiments are conducted to illustrate the advantage of the proposed middleware as well as the dynamic fault tolerance replication strategy. Comparison of the effectiveness of the proposed dynamic fault tolerance strategy and various traditional fault tolerance strategies are also provided.

1. Introduction

Web services are self-contained, self-describing, cross-platform and loosely-coupled computational components designed to support machine to machine interaction via networks, including the Internet. They are identified by Unified Resource Identifiers (URIs) and widely employed to implement the increasingly popular Service-Oriented Architectures/Applications (SOA). Employing the Extensible Markup Language (XML) for interface and message description, Web services provide unprecedented opportunities for building agile and versatile applications by integrating Web services provided by various service providers.

Reliability is a key issue of SOA which are widely employed in critical domains such as e-commerce and e-government. The highly dynamic nature of Web services and the Internet post a new challenge for application reliability improvement, which is not encountered in traditional stand-alone software. For example, Web services can be developed and published by anonymous developers without any quality guarantee, and may become unavailable easily. Moreover, the Internet traffic and server workload are also unpredictable, which will greatly influence the quality of Web services.

Fault tolerance is a major approach to achieve high reliability in traditional software reliability engineering [1]. Critics of software fault tolerance state that developing redundant software components for fault tolerance purpose is too expensive and the reliability improvement is questionable when comparing to a single system with all the effects of developing multiple redundant components. In the modern era of service-oriented computing, however, the cost of developing multiple version softwares and systems is greatly reduced. Because different companies compete with each other to provide their diversely designed/implemented Web services with identical or similar interfaces using various programming languages. Design diversity and cross-language implementation are therefore the natural outcome of Web services, making fault tolerance an attractive choice for SOA reliability enhancement. A number of static fault tolerance strategies for Web services have been proposed in the recent literature [2, 3, 4, 5, 6]. However, these static strategies are not feasible enough to be used in the highly dynamic Web environment. We need some "smart" replication strategies, which can be self-aware of the dynamic QoS changes of the target Web services and automatically determine the corresponding optimal fault tolerance strategy for service users.

Gaining inspiration from the user-participation and user-collaboration concepts of Web 2.0, we propose a user-participated QoS-aware middleware for dynamic optimal fault tolerance strategy determination. Our work aims at advancing the current state-of-the-art of fault tolerance in the field of service reliability engineering. The contributions of this paper include:

- **A QoS-aware middleware for achieving fault tol-**

1071-9458/08 $25.00 © 2008 IEEE
DOI 10.1109/ISSRE.2008.17

erance by employing user-participation and collaboration. By encouraging users to share their individually-obtained QoS information of the target Web services, more accurate evaluation on the available Web services can be achieved, which is important for the selection of the proper fault tolerance strategy with the right Web services.

- **A Dynamic fault tolerance strategy selection algorithm.** Comparing with the "static" fault tolerance strategies in the traditional software reliability engineering, more adaptable strategies are needed in the age of service-oriented computing, which is highly dynamic. We propose a dynamic fault tolerance strategy selection algorithm to determine the optimal strategy at runtime based on both the user requirements and the Web service QoS information.

The rest of this paper is organized as follows: Section 2 introduces our QoS-aware middleware. Section 3 proposes the dynamic fault tolerance strategy selection algorithm. Section 4 presents the experimental setup and experimental results, and Section 5 concludes the paper.

2. A QoS-Aware Middleware

In this section, some basic concepts are explained and the architecture of our QoS-aware middleware for fault tolerant Web services is presented. Our fault tolerance middleware can be integrated as one part of the SOA runtime governance [7].

2.1. Service Communities

With the popularization of service-oriented computing, various Web services are continuously emerging. The functionalities and interfaces enabled by Web Service Description Language (WSDL) are becoming more and more complex. Machine learning techniques [8, 9] are proposed to identify Web services with similar or identical interface automatically. However, the effect and accuracy of these approaches are still far from practical usage. Since identical services, which are provided by different providers, may appear with different function names, return types and parameter names, it is really difficult for machines to know that these services are actually providing the same service.

To solve the problem of identical/similar Web services identification, a service community defines a common terminology that is followed by all participants, so that the Web services, which are provided by different organizations, can be described in the same interface [10, 11]. Following a common terminology, easier resource aggregation can be achieved, which will help better development of the community.

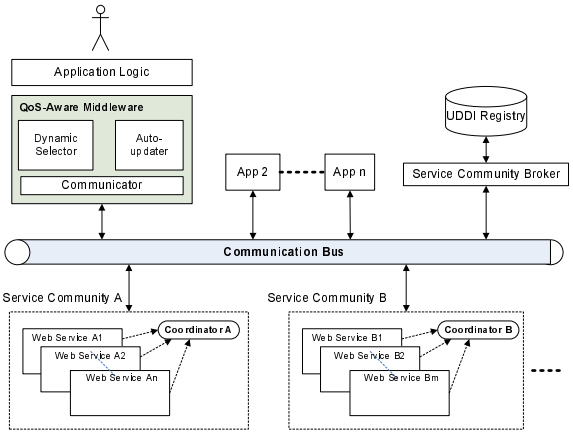

Figure 1. Architecture of the Middleware

Companies can enhance their business benefit by joining into communities. Since a lot of service users will go to the communities to search for suitable services. The coordinator of the community, which is a monitor application running on a stand-alone server, maintains a list of the registered Web services of the community. Before joining the community, a Web service has to follow the interface requirements of the community and registers at the community coordinator. By this way, the service community makes sure that various Web services in the community, which are provided by different organizations, come with the same interface.

In this paper, we focus on engaging the Web services in the same community for fault tolerance and performance enhancement purposes. We use the word *replica* to represent the Web services with identical interface within the same service community, which are provided by different organizations.

2.2. Architecture of the Middleware

The architecture of the proposed QoS-aware middleware for fault tolerant Web services is presented in Fig.1. The work procedure of this middleware is described as follows:

1. From the Universal Description, Discovery and Integration (UDDI), service users (usually service-oriented application developers) obtain the address of a particular service community coordinator.
2. By contacting the community coordinator, service users obtain an address list of the replicas in the community and the overall QoS information of these replicas. The overall QoS information will be used as the initial values in the middleware for optimal strategy selection. Detailed design of the QoS-model of Web services will be introduced in Sec.3.2.

3. Service users provide their particular QoS requirements to the middleware. Detailed design of the user requirement model will be introduced in Sec.3.1.
4. The proposed QoS-aware middleware selects an optimal fault tolerance strategy dynamically based on the user QoS requirements and the QoS information of target replicas.
5. The middleware invokes certain replicas with the selected optimal strategy and records down the QoS information of these replicas.
6. The middleware dynamically adjusts the optimal strategy based on the recorded replica QoS information.
7. As shown in Fig.2, in order to obtain the most up-to-date QoS information of target replicas for better strategy determination, the middleware will send its individually obtained replica QoS information to the community coordinator in exchange of the newest overall replica QoS information from time to time. In the other hand, by this QoS information exchange, the community coordinator can obtain replica QoS information from various service users in different geography locations, and use it for providing overall replica QoS information to service users.

As shown in Fig.1, the middleware includes the following three parts:

- **Dynamic selector:** In change of dynamically determining optimal fault tolerance strategy based on their requirements and the QoS information of replicas.
- **Auto updater:** Updating the newest overall replica QoS information from the community coordinator and providing the obtained QoS information to the coordinator. This mechanism promotes user collaboration to achieve more accurate optimal fault tolerance strategy selection.
- **Communicator:** In charge of invoking certain replicas with the selected optimal fault tolerance strategy.

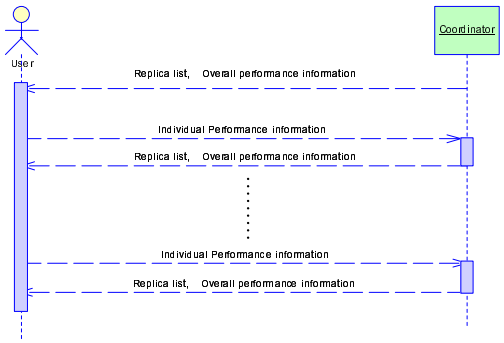

Figure 2. User and Coordinator Interaction

2.3. Types of Faults

Based on the cause of faults, various faults of Web service can be divided into the following two types:

- **Network-related faults.** Network-related faults are generic to all Web services. For example, *Communication Timeout, Service Unavailable (http 503), Bad Gateway (http 502), Server Error (http 500)*, and so on, are network-related faults. They can be easily identified by the judgement function in the middleware.
- **Logic-related faults.** Logic-related faults are specific to different Web services. For example, Web service returning the wrong result, calculation faults, data faults, and so on, are logic-related faults. Also, various exceptions thrown by the Web service to the service users are classified into the logic-related faults. It is difficult for the middleware to identify such type of faults

2.4. Fault Tolerance Strategies

When applying Web services to critical domains, reliability becomes a major issue. With the popularization of Web services and service communities, more and more Web services with an identical interface are diversely designed and implemented by different organizations, making service-level fault tolerance a attractive choice for service reliability improvement.

Retry [4] and *Recovery Block (RB)* [12] are two major sequential approaches that use time redundancy to obtain higher reliability. They have been employed in FT-SOAP [13] and FT-CORBA [14]. In the other hand, *N-Version Programming (NVP)* [15] and *Active* [8] strategies are two major parallel strategies that engage space/resource redundancy for reliability improvement. They have been employed in FTWeb [16], Thema [17], WS-Replication [18] and in [19].

In the following, we provide detailed introduction and the formulas of response time and failure-rate of these traditional fault tolerance strategies. As discussed in the work [20], we assume that each request is independent, and the Web service fails at a fix rate. Here, we use RTT (Round-Trip-Time) to represent the time duration between service user sending out a request and receiving a response.

- **Retry:** The same Web service will be retried for a certain number of times when it fails. Eq.1 is the formula for calculating failure-rate f and RTT t, where m is the retry times, f_1 is the failure-rate of the target Web service, and t_i is the RTT of the i^{th} request.

$$f = f_1^m; \qquad t = \sum_{i=1}^{m} t_i (f_1)^{i-1} \tag{1}$$

- **RB:** Another standby Web service will be tried sequentially if the primary Web service fails.

$$f = \prod_{i=1}^{m} f_i; \qquad t = \sum_{i=1}^{m} t_i \prod_{k=1}^{i-1} f_k \tag{2}$$

- **NVP:** NVP invokes different replicas at the same time and determines the final result by majority voting. It is usually employed to mask logical faults. In Eq.3, v, which is an odd number, represents the total replica number. $F(i)$ represents the failure-rate that i ($i \leq v$) replicas fail. For example, assuming $v = 3$, then $f = \sum_{i=2}^{3} F(i) = F(2)+F(3) = f_1 \times f_2 \times (1-f_3)+f_2 \times f_3 \times (1-f_1)+f_1 \times f_3 \times (1-f_2)+f_1 \times f_2 \times f_3$.

$$f = \sum_{i=v/2+1}^{v} F(i); \qquad t = \max(\{t_i\}_{i=1}^{v}) \tag{3}$$

- **Active:** $Active$ strategy invokes different replicas in parallel and takes the first properly-returned response as the final result. It is usually employed to mask network faults and to obtain better response time performance. In Eq.4, T_c is a set of RTTs of the properly-returned responses. u is the parallel replica number.

$$f = \prod_{i=1}^{u} f_i; t = \begin{cases} \min(T_c) : |T_c| > 0 \\ \max(T) : |T_c| = 0 \end{cases} \tag{4}$$

The highly dynamic nature of Web services makes the above static fault tolerance strategies unpractical in real-world environment. For example, some replicas may become unavailable permanently, while some new replicas may join in. Moreover, Web service software/hardware may be updated without any notification, and the Internet traffic load and service server workload are also changing from time to time. These unpredictable characteristics of Web services provide a challenge for optimal fault tolerance strategy selection. To address this challenge, we propose the following two dynamic fault tolerance strategies, which are more adaptable and can be automatically determined by the QoS-aware middleware in runtime. These two dynamic strategies will be employed in our dynamic fault tolerance strategy selection algorithm in Sec.3.4.

- **Dynamic Sequential Strategy:** The dynamic sequential strategy is the combination of $Retry$ and RB strategies. When the primary replica fails, our algorithm will determine whether to employ $Retry$ or RB dynamically at runtime based on the QoS of target replicas and the requirements of service users. The determining algorithm will be introduced in Sec.3.4. In Eq.5, m_i is the retry times of the i^{th} replica, and n is the total replica quantity. This strategy equals RB when $m_i = 1$, and equals to $Retry$ when $m_1 = \infty$.

$$f = \prod_{i=1}^{n} f_i^{m_i}; t = \sum_{i=1}^{n} ((\sum_{j=1}^{m_i} t_i f_i^{j-1}) \prod_{k=1}^{i-1} f_k^{m_i}) \tag{5}$$

- **Dynamic Parallel Strategy:** The dynamic parallel strategy is the combination of NVP and $Active$. It will invoke u replica at the same time and employ the first v (v is an odd number, and $v \leq u$) properly-returned responses for majority voting. This strategy equals to $Active$ when $v = 1$, and equals to NVP when $v = u$. $middle(v, T_c)$ is employed to calculate the RTT of invoking u replica in parallel and including the first v for voting, which is equal to the RTT of the v^{th} properly-returned response.

$$f = \sum_{i=v/2+1}^{v} F(i); t = \begin{cases} middle(v, T_c) : |T_c| \geq v \\ \max(T) : |T_c| < v \end{cases} \tag{6}$$

3. Dynamic Fault Tolerance Strategy Selection Algorithm

3.1. User Requirement Model

Optimal fault tolerance strategies for service-oriented applications vary from case to case, which are influenced not only by QoS of target replicas, but also by the characteristics of service-oriented applications. For example, latency-sensitive applications may prefer parallel strategies for better response time performance, while resource-constrained applications may employ sequential strategies for better resource conservation.

It is generally difficult for a middleware to automatically detect the characteristics of the service-oriented application, such as whether it is latency-sensitive or resource constrained. The strategy selection accuracy will be greatly enhanced if the service users can provide some concrete requirement information. However, it is impractical and not user-friendly to require the service users, who are often not familiar with fault tolerance strategies to provide detailed technical information. To address this problem, we design a simple user requirement model for obtaining necessary requirement information from the users. In this model, the users are required to provide the following four values:

1. t_{max}: the largest RTT that the application can afford. t_{max} with a smaller value means higher requirement on RTT, indicating the realtime characteristic of the application. RTT larger than t_{max} will be considered as timeout for the service users.
2. f_{max}: the largest failure-rate that the application can tolerate.

3. r_{max}: the largest resource consumption constraint. The amount of parallel connection is used to approximately quantitate the resource consumption, since connecting more Web services in parallel will consume more computing and networking resources. r_{max} with a smaller value indicates that the application is resource-constraint.
4. $mode$: the $mode$ can be set by the service users to be $sequential$, $parallel$, or $auto$. $Sequential$ means invoking the replicas sequentially. For example, it is more suitable to visit payment-oriented Web services sequentially. We need the service users to provide this $mode$ information, because the middleware may not be smart enough to detect whether the target replicas are payment-oriented services or not. $Parallel$ means that the user prefers invoking the target replicas in parallel. $Auto$ means that the users let the middleware determine the optimal mode automatically.

The user requirement information obtained by this model will be used in our dynamic fault tolerance strategy selection algorithm in Sec.3.4.

3.2. QoS Model of Web Service

In addition to the subjective user requirements, the objective QoS information of the target Web service replicas are also needed for the optimal fault tolerance strategy selection. A lot of previous tasks are focused on building the QoS model for Web services [21, 22, 23]. However, there are still several challenges to be solved:

- **It is difficult to obtain performance information of target Web services.** Service users do not always record the QoS information of target replicas, such as RTT, failure-rate and so on. Also, most of the service users are unwilling to share the QoS information they obtain.
- **Distributed geography location of users make evaluation on target Web services difficult.** Web service performance is highly related to the geography location of both the service user and the service provider, making performance evaluation results provided by an individual user easy to be misinterpreted by others across the Web. For example, a user located in the same local area network (LAN) with the target Web service is more likely to obtain very good performance. The optimistic evaluation result provided by this user may misguide other users who are not in the same LAN with the target Web service.
- **Lack of a convenient mechanism for service users to obtain QoS information of Web services.** QoS information can help service users be aware of the quality of a certain Web service and determine whether to use it or not. However, in reality, it is very difficult for the service users to obtain accurate and objective QoS information of the Web services.

To address the above challenges, we design a QoS model of Web services employing the concept of user-participation and user-collaboration, which is the key innovation of Web2.0. The basic idea is: by encouraging users to contribute their individual obtained QoS information of target replicas, we can collect a lot of QoS data from the users located in different geography locations under various network conditions, and engage these data to make objective overall evaluation on the target Web services.

Based on the concept of service community and the architecture designed in Fig.1, we use the community coordinator to store the QoS information of the replicas. Users will periodically send their individually-obtained replica QoS information to the service community in exchange of the the newest overall replica QoS information, which can be engaged for better optimal strategy selection. Since the middleware will record replica QoS data and exchange it with the coordinator automatically, updated replica QoS information is conveniently available for service users.

For a single replica, the community coordinator will store the following information:

- t_{avg} : the average RTT of the target replica.
- t_{std} : the standard deviation of RTT of the target replica.
- fl : the logic failure-rate of the target replica.
- fn : the network failure-rate of the target replica.

To simplify the model, we only consider the most important QoS properties, including RTT, logic faults, network faults and resource consumption. Other QoS properties, however, can be easily included in the future. For those users who are not willing to exchange QoS data with the community coordinator, they can simply close the exchange functionality of the middleware, although this will reduce the dynamic optimal strategy selection performance. This is similar to BitTorrent [24] download, where stopping uploading files to others will hurt the download speed of the user.

3.3. A Scalable RTT Prediction Algorithm

Accurate RTT prediction is important for optimal fault tolerance strategy selection. Assuming, for example, that there are totally n replicas $\{ws\}_{i=1}^{n}$ in the service community. We would like to invoke v ($v \leq n$) replicas in parallel and use the first properly-returned response as the final result. The question is, then, how to find out the optimal set of replicas that will achieve the best RTT performance?

To solve this problem, we need the RTT distributions of all the replicas. In our previous work [5], all the historical RTT results are stored and employed for RTT performance prediction. However, it is impractical to require the users to store all the past RTT results, which are ever growing and will consume a lot of storage memory. On the other hand, without historical RTT performance information of the replicas, it is extremely difficult to make accurate prediction.

We design the following approach for obtaining the RTT distributions of a certain replica, which scatters the RTT distributions, and reduces the required data storage.

We divide the time t_{max}, which is provided by the service user, into k time slots. Instead of storing all the detailed past RTT results, the service user only needs to store $k+2$ counters $\{c_i\}_{i=1}^{k+2}$ for a replica, where k counters of which are used to record the RTT numbers of the corresponding time slots, and another two counters are used to record network-related faults fn and logic-related faults fl, respectively. Employing these counters, Eq.7 can be used to predict the probability that a RTT of a certain replica belonging to a certain category.

$$p_i = \frac{c_i}{\sum_{i=1}^{k+2} c_i} \tag{7}$$

By the above design, we can obtain approximate RTT distribution information of a replica by storing only $k+2$ counters. Also, the time slot number k can be set to be a larger value for obtaining more detailed distribution information, making this algorithm scalable.

The approximate RTT distributions of the replicas, which are obtained by the above approach, can be engaged to predict RTT performance of a particular set of replicas $\{ws_i\}_{i=1}^{v}$. The problem of predicting RTT performance can be formulated as follows:

Problem 1 *Given:*

- $\{ws_i\}_{i=1}^{v}$: a set of target replicas for prediction.
- $\{p_{i,j}\}_{j=1}^{k+2}$: for replica i ($1 \leq i \leq v$), the probability of an RTT belonging to different categories.
- $\{t_i\}_{i=1}^{k}$: the RTT value of the time slot i, which can be calculated by $t_i = (t_{max} \times i)/k - t_{max}/(2 \times k)$.
- $T_v = \{rtt_j\}_{j=1}^{v}$: a set of RTT of the v replicas, where the probability of rtt_j belonging to the time slot k is provided by $p_{j,k}$.

Find out:

- $\mathrm{E}(\min(T_v))$: the average response time by invoking all the v replicas in parallel for many times, where function $\min(T_v)$ stands for the minimal RTT value of all the $\{rtt_j\}_{j=1}^{v}$.

By employing Eq.8,

$$\mathrm{E}(\min(T_v)) = \sum_{i=1}^{k} (\mathrm{P}(\min(T_v) == t_i) \times t_i), \tag{8}$$

the problem of finding $\mathrm{E}(\min(T_v))$ can be transferred to finding $\mathrm{P}(\min(T_v) == t_i)$, which means the probability of the minimal RTT value of T_v belonging to the time slot i. Since $\mathrm{P}(\min(T_v) == t_i) = \mathrm{P}(\min(T_v) \leq t_i) - \mathrm{P}(\min(T_v) \leq t_{i-1})$, the problem becomes to $\mathrm{P}(\min(T_v) \leq t_i)$. If one of the v replicas, for example rtt_n, in T_v is smaller than t_i, then $\min(T_v)$ will be smaller than t_i. Therefore $\mathrm{P}(\min(T_v) \leq t_i) = \mathrm{P}(rtt_n \leq t_i) + \mathrm{P}(rtt_n > t_i) \times \mathrm{P}(\min(T_{v-1}) \leq t_i)$. The probability of $rtt_i \leq t_j$ can be calculated by Eq.9:

$$\mathrm{P}(rtt_i \leq t_j) = \sum_{k=1}^{j} p_{i,k}. \tag{9}$$

Therefore, by the above calculation, the RTT performance of the *Active* strategy, which invokes the given replicas in parallel and employs the first returned response as final result, can be predicted. By changing the the function $\min(T_v)$ to $\max(T_v)$, the above algorithm can be used to predict the RTT performance of the NVP strategy, which needs to wait for all responses of replicas before voting. By changing the function $\min(T_v)$ to $middle(T_v, y)$, which means the RTT value of the y^{th} returned response, the above algorithm can be used to predict the RTT performance of the *Dynamic parallel strategy*. For example, in the *Dynamic parallel strategy*, we invoke 6 replicas in parallel and employ the first 3 returned response for voting, the overall RTT performance will be equal to the RTT of the 3^{rd} returned response.

3.4. A Dynamic Fault Tolerance Strategy Selection Algorithm

Employing the user requirement model designed in Sec.3.1, the QoS model of Web services designed in Sec.3.2, and the RTT prediction algorithm designed in Sec.3.3, we propose a dynamic fault tolerance strategy selection algorithm in this section. The whole selection procedure is composed of three parts: sequential or parallel strategies determination, dynamic sequential strategy determination, and dynamic parallel strategy determination.

3.4.1 Sequential or Parallel Strategy Determination

If the value of the attribute *mode* in the user requirement model equals to *auto*, we need to conduct sequential or parallel strategy determination based on the QoS performance of the target replicas and subjective requirements of

the users. Eq.10 is used to calculate the performance of different strategies:

$$p_i = \frac{t_i}{t_{max}} + \frac{f_i}{f_{max}} + \frac{r_i}{r_{max}}. \quad (10)$$

The underlying consideration is that the performance of a particular response time is related to the user requirement. For example, 100 ms is a large latency for the latency-sensitive applications, while it may be neglected for non-latency-sensitive applications. By using $\frac{t_i}{t_{max}}$, where t_{max} represents the user requirement on response time, we can have a better representation of the response time performance for service users with different requirements. Failure-rate f_i and resource consumption r_i are similarly considered.

By employing Eq.10, the performance of sequential strategies and parallel strategies can be computed. For sequential strategies, the value of t_i can be calculated by Eq.5, where the value of f_i can be obtained from the middleware and the value of r_i is 1 (only one replica is invoked at the same time). For parallel strategies, the value of t_i can be estimated by using the RTT prediction algorithm presented in Sec.3.3, where the value of f_i can be obtained from the middleware, and the value of r_i is the number of parallel invocation replicas.

The performance results of the sequential and parallel strategies, which are obtained by the above procedure, are used for determining whether to use sequential or parallel strategies.

3.4.2 Dynamic Sequential Strategy Determination

If the value of the attribute $mode$ provided by the service user is equal to $sequential$, or the sequential strategy is selected by the above selection procedure conducted by the middleware, we need to determine the detailed sequential strategy dynamically based on the user requirements and replica QoS information.

$d = \frac{1}{m} \times (\frac{t_{i+1}-t_i}{t_{max}} + \frac{f_{i+1}-f_i}{f_{max}})$ is used to calculate the performance difference between two replicas, where m is the retry times. When $d > a$, where a is the performance degradation threshold, the performance difference between the two selected replicas is large, therefore, retrying the original replica is more likely to obtain better performance. With increasing the retry times m, d will become smaller and smaller, reducing the priority of strategy $Retry$.

If the primary replica fails, the above procedure will be repeated until either a success or the time is out ($RTT \geq t_{max}$).

3.4.3 Dynamic Parallel Strategy Determination

If the value of the attribute $mode$ provided by the service user is equal to $parallel$, or the parallel strategy is selected by the middleware, we need to determine the optimal parallel replica number n and the NVP number v ($v \leq n$) for the dynamic parallel strategy.

By employing the RTT prediction algorithm presented in Sec.3.3, we can predict the RTT performance of various combinations of the value v and n. The number of all combinations can be calculated by $C_n^v = \frac{n!}{v! \times (n-v)!}$, and the failure-rate can be calculated by using Eq.6. By employing Eq.10, the performance of different n and v combination can be calculated and compared. The combination with the minimal p value will be selected and employed as the optimal strategy.

4. Experiments

A series of experiments is designed and performed for illustrating the QoS-aware middleware and the dynamic fault tolerance selection algorithm. In the experiments, we compare the performance of our dynamic fault tolerance strategy $Dynamic$ with other four traditional fault tolerance strategies $Retry$, RB, NVP, and $Active$.

4.1. Experimental Setup

Our experimental system is implemented and deployed with JDK6.0, Eclipse3.3, Axis2.0 [25], and Tomcat6.0. We develop six Web services following an identical interface to simulate replicas in a service community. These replicas are employed for evaluating the performance of various fault tolerance strategies under different situations. The service community coordinator is implemented by *Java Servlet*. The six Web services and the community coordinator are deployed on seven PCs. All PCs have the same configuration: Pentium(R) 4 CPU 2.8 GHz, 1G RAM, 100Mbits/sec Ethernet card, and a Windows XP operating system.

In the experiments, we simulate network-related faults and logic-related faults. All the faults are further divided into permanent faults (service is down permanently) and temporary faults (faults occur randomly). The fault injection techniques are similar to the ones proposed in [26, 27].

In our experimental system, service users, who will invoke the six Web service replicas, are implemented as *Java applications*. We provide six service users with representative requirement settings as typical examples for investigating performance of different fault tolerance strategies in different situations. The detailed user requirements, which are obtained by the user requirement model proposed in Sec.3.1, are shown in Table 1. In the experiments, failures are counted when service users, who employ a certain type of fault tolerance strategy, cannot get a proper response. For each service request, if the response time is larger than t_{max}, a $timeout$ failure is counted, whatever the reason leading to the late response.

Table 1. Requirements of Service Users

Users	t_{max}	f_{max}	r_{max}	Focus
User 1	1000	0.1	50	RTT
User 2	2000	0.01	20	RTT, Fail
User 3	4000	0.03	2	RTT, Fail, Res
User 4	10000	0.02	1	Res
User 5	15000	0.005	3	Fail, Res
User 6	20000	0.0001	80	Fail

Table 2. Parameters of Experiments

	Parameters	Setting
1	Number of replicas	6
2	Network fault probability	0.01
3	Logic fault probability	0.0025
4	Permanent fault probability	0.05
5	Number of time slots	20
6	Performance degradation threshold (a)	2
7	Replica number of NVP	5
8	Parallel replica number of $Active$	6
9	Dynamic degree	20

Our experimental environment is defined by a set of parameters, which are shown in Tables 2. The *permanent fault probability* means the probability of permanent faults among all the faults, which includes *network-related faults* and *logic-related faults*. The *performance degradation threshold* is employed by the dynamic strategy selection algorithm, which has been introduced in Sec.3.4. *Dynamic degree* is used to control the QoS changing of replicas in our experimental system, where a larger number means more serious changing of QoS properties, such as the RTT.

4.2. Experimental Results

The experimental results of the six service users employing different types of fault tolerance strategies are shown in Table 3-8. The results include the employed fault tolerance strategy (*Strategies*), the number of all requests (*All*), the average RTT of all requests (*RTT*), the number of failure (*Fail*), the average consumed resource (*Res*), and the overall performance (*Perf*, calculated by Eq. 10). The time units of RTT is in milliseconds (ms).

In the following, we provide detailed explanation on the experimental results of Service User 1.

As shown in Table 1, the requirements provided by User 1 are: $t_{max} = 1000$, $f_{maz} = 0.1$ and $r_{max} = 50$. These requirement settings indicate that User 1 cares more on the response time than the failure-rate and resources, because 1000 ms maximal response time setting is tight in the high dynamic Internet environment, and the settings of failure-rate and resource consumption are loose. As shown in Table 3, among all the strategies, the RTT performance of the strategy NVP is the worst, while the RTT performance of the strategy $Active$ is the best. This is reasonable, since NVP needs to wait for all responses before majority voting, while $Active$ simply employs the first properly-returned response as the final result. We can see that the proposed $Dynamic$ strategy can provide good RTT performance to User 1.

Table 3. Experimental Results of User 1

U	Strategies	All	RTT	Fail	Res	Perf
1	Retry	50000	420	2853	1	1.011
	RB	50000	420	2808	1	1.002
	NVP	50000	839	2	5	0.939
	Active	50000	251	110	6	0.393
	Dynamic	50000	266	298	2.34	0.372

The $Fail$ column in Table 3 shows fault tolerance performance of different strategies. The failure-rates of the strategies $Retry$ and RB are not good, because the setting of $t_{max} = 1000ms$ leads to a lot of timeout failures. Among all the strategies, NVP obtains the best fault tolerance performance. This is not only because NVP can tolerate logic-related faults by majority voting, but also because NVP invokes 5 replicas in parallel in our experiments, which greatly reduces the number of $timeout$ failures. For example, if one replica does not respond within the required time period t_{max}, NVP can still get the correct result by conducting majority voting using the remaining responses. The fault tolerance performance of the $Dynamic$ strategy is not good comparing with NVP. However, this fault tolerance performance is already good enough for User 1, who does not care so much about the failure-rate ($f_{max} = 0.1$).

The Res column in Table 3 shows the resource consumption information of different fault tolerance strategies. We can see that the resource consumption of strategies $Retry$ and RB is equal to 1, because these two strategies invoke only one replica at the same time. As shown in Table 2, the version number of strategy NVP is set to be 5 and the parallel invocation number of strategy $Active$ is set to be 6 in our experiments; therefore, the Res of these two strategies

Table 4. Experimental Results of User 2

U	Strategies	All	RTT	Fail	Res	Perf
2	Retry	50000	471	285	1	5.985
	RB	50000	469	283	1	5.944
	NVP	50000	855	0	5	0.677
	Active	50000	253	126	6	2.946
	Dynamic	50000	395	3	4.03	0.459

Table 5. Experimental Results of User 3

U	Strategies	All	RTT	Fail	Res	Perf
3	Retry	50000	458	155	1	0.717
	RB	50000	457	149	1	0.713
	NVP	50000	845	1	5	2.712
	Active	50000	248	138	6	3.154
	Dynamic	50000	456	141	1	0.708

Table 6. Experimental Results of User 4

U	Strategies	All	RTT	Fail	Res	Perf
4	Retry	50000	498	145	1	1.194
	RB	50000	493	131	1	1.180
	NVP	50000	868	1	5	5.087
	Active	50000	251	119	6	6.144
	Dynamic	50000	494	109	1	1.158

Table 7. Experimental Results of User 5

U	Strategies	All	RTT	Fail	Res	Perf
5	Retry	50000	454	115	1	0.823
	RB	50000	450	121	1	0.847
	NVP	50000	779	0	5	1.718
	Active	50000	249	125	6	2.516
	Dynamic	50000	489	60	1.46	0.759

Table 8. Experimental Results of User 6

U	Strategies	All	RTT	Fail	Res	Perf
6	Retry	50000	470	146	1	29.236
	RB	50000	468	119	1	23.835
	NVP	50000	839	1	5	0.304
	Active	50000	249	132	6	26.487
	Dynamic	50000	473	1	3.56	0.268

are 5 and 6, respectively. The *Dynamic* strategy invokes 2.34 replicas in parallel on average.

From the *Perf* column shown in the Table 3, we can see that *Dynamic* strategy achieves the best overall performance (calculated by Eq.10) among all the strategies. Although the *Active* strategy also achieves good performance for User 1, it is not good to other users with different requirement.

As shown in Tables 4-8, for other service users, the *Dynamic* strategy can also provide suitable strategy dynamically to achieve good performance. As shown in Fig.3, the *Dynamic* strategy provides the best overall performance among all the fault tolerance strategies for all the six service users. This is because the *Dynamic* strategy considers user requirements and can adjust itself for optimal strategy dynamically according to the change of replica QoS. The other four traditional fault tolerance strategies

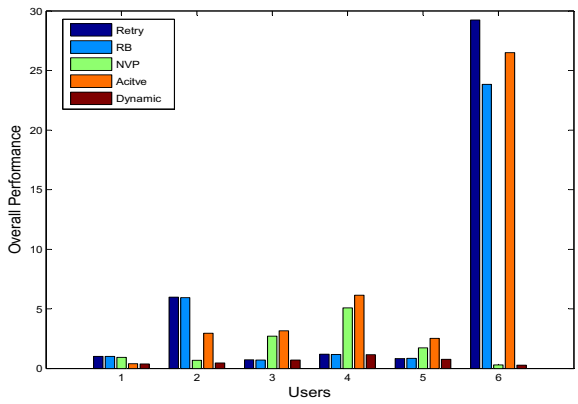

Figure 3. Overall Performance of Strategies

perform well in some situations; however, they perform badly in other situations, because they are too static. Our experimental results indicate that the traditional fault tolerance strategies may not be good choices in the field of service-oriented computing, which is highly dynamic. The experimental results also indicate that our proposed *Dynamic* fault tolerance strategy is more adaptable and can achieve better overall performance comparing with the traditional fault tolerance strategies.

5. Conclusion

This paper proposes a QoS-aware middleware for fault tolerant Web services. Based on this middleware, service users share their individually-obtained Web service QoS information with each other via a service community coordinator. A dynamic optimal strategy selection algorithm, which employs both objective replica QoS information as well as subjective user requirements, is designed and evaluated. Experiments are conducted and the experimental results indicate that the proposed *Dynamic* strategy can obtain better overall performance for various service users comparing with the traditional fault tolerance strategies.

Currently, we only consider the most important QoS properties (RTT, failure-rate, and resources) in the middleware. More QoS properties will be involved in the future. Also, the proposed fault tolerance middleware can only work on stateless Web services at the current stage. More investigations are needed for the fault tolerance of stateful Web services.

6. Acknowledgement

The work described in this paper was fully supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK4150/07E), and a grant from the Research Committee of The Chinese University of Hong Kong (Project No. CUHK3/06C-SF).

References

[1] M. Lyu (ed.), "Handbook of Software Reliability Engineering," McGraw-Hill, New York, 1996.

[2] W.T. Tsai, R Paul, L Yu, A Saimi, Z Cao, "Scenario-Based Web Service Testing with Distributed Agents," *IEICE Transaction on Information and System*, Vol. E86-D, No.10, pp. 2130-2144, 2003.

[3] H. Foster, S. Uchitel, J. Magee, and J. Kramer, "Model-based Verification of Web Service Compositions", *in 18th IEEE International Conference on Automated Software Engineering*, pp.152-161, 2003.

[4] P. W. Chan, M.R. Lyu and M. Malek, "Reliable Web Services: Methodology, Experiment and Modeling," *in IEEE International Conference on Web Services*, pp. 679-686, 2007.

[5] Z. Zheng, M.R. Lyu, "WS-DREAM: A Distributed Reliability Assessment Mechanism for Web Services", *in 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'08)*, Anchorage, Alaska, USA, June 24-27, 2008.

[6] Z. Zheng, M.R. Lyu, "A Distributed Replication Strategy Evaluation and Selection Framework for Fault Tolerant Web Services", in Proceedings of the IEEE International Conference on Web Services (ICWS'08), Beijing, China, Sep. 23-26, 2008.

[7] M. Kavianpour, "SOA and Large Scale and Complex Enterprise Transformation", in Proceedings of the Fifth International Conference on Service-Oriented Computing (ICSOC'07), Vienna, Austria, Sep. 17-20, 2007.

[8] N. Salatge and J.C. Fabre, "Fault Tolerance Connectors for Unreliable Web Services," *in 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, Edinburgh, UK, pp. 51-60, 2007.

[9] J. Wu and Z. Wu, "Similarity-based Web Service Matchmaking," *in IEEE International Conference on Services Computing (SCC'05)*, Vol. 1, pp. 287-294, 2005.

[10] B. Benatallah, M. Dumas, Q. Z. Sheng and A. H. H. Ngu, "Declarative Composition and Peer-to-Peer Provisioning of Dynamic Web Services," *in Proceedings of the 18th International Conference on Data Engineering (ICDE 2002)*, 2002, pp. 297-308.

[11] L. Zeng, Boualem Benatallah, Anne H. H. Ngu, Marlon Dumas, Jayant Kalagnanam, Henry Chang, "QoS-Aware Middleware for Web Services Composition", *IEEE Transaction on Software Engineering*, vol. 30, no. 5, pp. 311-327, 2004.

[12] B. Randell and J. Xu, "The Evolution of the Recovery Block Concept," Software Fault Tolerance, M. R. Lyu (ed.), Wiley, Chichester, 1995, pp. 1-21.

[13] D. Liang, C. Fang, and C. Chen, "FT-SOAP: A Fault Tolerant Web Service," *in Tenth Asia-Pacific Software Engineering Conference*, Chiang Mai, Thailand, 2003.

[14] D. Liang, C. Fang and S. Yuan, "A Fault-Tolerant Object Service on CORBA," *Journal of Systems and Software*, Vol. 48, pp. 197-211, 1999.

[15] A.A. Avizienis, "The Methodology of N-Version Programming," Software Fault Tolerance, M. R. Lyu (ed.), Wiley, Chichester, 1995, pp. 23-46.

[16] G. T. Santos, L. C. Lung, and C. Montez, "FTWeb: A Fault Tolerant Infrastructure for Web Services," *in Ninth IEEE International EDOC Enterprise Computing Conference (EDOC'05)*, pp. 95-105, September 2005.

[17] M. G. Merideth, A. Iyengar, T. Mikalsen, S. Tai, I. Rouvellou, and P. Narasimhan, "Thema: Byzantine-Fault Tolerant Middleware for Web-Service Applications", *in IEEE Symposium on Reliable Distributed Systems*, pp. 131-140, 2005.

[18] J. Salas, F. Perez-Sorrosal, M. Patino-Martinez, and R. Jimenez-Peris, "WS-Replication: a Framework for Highly Available Web Services", *in 15th International Conference on the World Wide Web*, pp. 357-366, 2006.

[19] J. Osrael, L. Froihofer, K. M. Goeschka, S. Beyer, P. Galdamez, and F. Munoz, "A System Architecture for Enhanced Availability of Tightly Coupled Distributed Systems", *in 1st International Conference on Availability, Reliability and Security (ARES06)*, pp. 400C407, 2006.

[20] D. Leu, F. Bastani and E. Leiss, "The Effect of Statically and Dynamically Replicated Components on System Reliability", *IEEE Transactions on Reliability*, vol.39, issue 2, pp. 209-216, June 1990.

[21] E.M. Maximilien, and M.P.Singh, "Conceptual Model of Web Service Reputation", *in ACM SIGMOD Record (Special section on semantic web and data management)*, vol.31, issue 4, pp. 36-41, 2002.

[22] G. Wu, J. Wei, X. Qiao and L. Li, "A Bayesian Network based QoS Assessment Model for Web Services", *in IEEE International Conference on Services Computing (SCC 2007)*, Utah, USA, pp. 498-505, 2007.

[23] V. Deora, J.H. Shao, W.A.Gray and N.J.Fiddian, "A Quality of Service Management Framework based on User Expectations", *in First International Conference on Service Oriented Computing (ICSOC 2003)*, Trento, Italy, 2003.

[24] BitTorrent, http://www.bittorrent.com

[25] Apache Axis2, http://ws.apache.org/axis2

[26] N. Looker and J. Xu, "Assessing the Dependability of Soaprpc- based Web Services by Fault Injection," *in Proc. of the 9th IEEE International Workshop on Object-oriented Real-time Dependable Systems*, pp. 163-170, 2003.

[27] M. Vieira, N. Laranjeiro, and He. Madeira, "Assessing Robustness of Web-Services Infrastructures," *in 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, Edinburgh, UK, pp. 131-136, 2007.